

1 Main Notes

Contents

1	Main Notes	1
2	Fault Tolerant Systems	2
2.1	Introduction	2
2.2	Examples	2
2.3	Reliability, Availability and Related Concepts	2
2.4	Modelling Reliability	3
2.5	How to make a system more fault tolerant	4
2.5.1	Redundancy in Hardware (Components)	4
2.5.2	Redundancy in Software	5
2.5.3	Redundancy in Information	5
3	Sensors	6
3.1	Introduction	6

2 Fault Tolerant Systems

2.1 Introduction

So, what does it mean for a system to be fault tolerant, what does it mean for a system to be reliable? How do we know if a system is reliable? How can we make it more reliable? These concepts will be explored in this chapter.



Definition — Fault Tolerance

A system is fault tolerant if

the system continues to operate CORRECTLY and SAFELY in the presence of faults

Simple enough, but what is a fault?



Definition — Fault

A fault is an unexpected irregularity within the input or output of the system.

We should note that it is impossible to make something completely fault proof, there are too many environmental factors to make this possible. With these basic definitions under our belt, we will explore some systems that proved not fault tolerant, and some that proved to be and why.

But first, let's list some particular industries where fault tolerance is not a nice-to-have, it is a necessity. Then we will look at specific examples.

2.2 Examples

1. Transportation

Can be broken down further into specific industries like cars, boats, trains, planes, etc

The flops

Big flop is Boeing. The issue was that they had made a new derivative of the 737, the 737 MAX. Boeing had made several changes to the 737 MAX, and one of these changes was a system to warn the pilots if the plane was stalling using an angle of attack sensor. The 737 MAX had two, however the software system only used one, and never checked the measured values against other sensors that could be used to complement the reading for accuracy. Furthermore, because pilots were never told about this new feature, they were not ready to handle it. In short, the fleet was grounded because the new system had a **single point of failure**. And when this went, everything else did too.

2. Banking

3. Medical Devices

4. Space

5. Energy

6. Communications

2.3 Reliability, Availability and Related Concepts



Definition — Reliability

Reliability has a specific definition for us. While we know what it MEANS to be reliable or not reliable, reliability is:

the probability that a system will continue operation correctly during a period of time



When we say correctly, we mean it!

Note that this is for continuous reliability, if you say that system has 0.99 percent reliability during a 5 hour interval, there cannot be these consistent spikes in instability after the first hour, that means the system is not 0.99 percent reliable during this time period.



Definition — Availability

Availability is defined as:

the probability that at a given moment, the system is working correctly and is available.

Note that availability and reliability are separate concepts. A system can be reliable without being available. For example, take a hypothetical car. This car, when it detects a hazard is about to cause a crash, it will take controls away from the driver and autopilot the car to safety. During such a moment, the steering, gas and brakes did not fail, they are working exactly as intended. But they are not available to the driver! A simpler example could be like scheduled maintenance. The server is down, but that is the point, maintenance is ongoing.



Definition — Mean Time To Fail – MTTF

How long on average does it take for a system to fail?

2.4 Modelling Reliability

As with most things, engineers and scientists cannot resist modelling stuff with equations. Reliability is no exception. We generally say that reliability is modelled as:

$$R(t) = e^{-\lambda t} \quad (1)$$

where t is defined as the time, and the output is the reliability of the system at t . So if we wanted our .99 reliability eat 5 hours, we would need for λ to be somewhere around 0.002. Ok lambda is great and all, but what does this really mean? How do we make λ 0.002? Can an AMD CPU be rated at $\lambda = 0.002$? So turns out, we have another, more graphical, system to help model reliability.



Why we use an exponential function to model reliability?

We use an exponential function with a negative power because intuitively, as a system runs for a longer period of time, the less reliable it becomes.



Concept — Success Tree

A success tree is a tree that one can build using circuit/electrical symbol diagrams that is intended to be used to model the reliability of a given system.

We can borrow from circuitry/electrical diagram design to make our model. So lets say we have a computer and a battery. We want these things to run together because they are part of the system (computer does the computer, battery provides the power, whatever). So the computer and battery are illustrated however we want. But then we connect them via an AND gate to show that they have to work together. At the outset, each component we give it some numeric value to show how reliable it is. A quality battery from Duracell might get a 0.98, while the one you found on Temu might get a 0.20. To model the reliability of the system with the computer and the battery, you will multiply their reliability together.

This cl

So that is running two things together at the same time, but what about if we can have two things, but are only used for redundancy? So one is running the other isn't. So in our previous example, we might have two batteries, one for primary power, the second as an auxiliary power source if the primary fails. For this, we use an OR gate to illustrate this. And, instead of multiplying, we take the reliability of both components, subtract it from 1 to get the probability of failure, then multiply those two together, then subtracting the product from 1. Now our reliability goes up instead of down, genius!

So then it would seem that the solution to everything is to just make everything infinitely redundant, and that way nothing ever fails. While that would be true to some extent, this is not always possible. For example if you are landing a rover on Mars, you first need to get that thing into space. Weight becomes a constraint. The heavier your lander is, the more fuel you need to put it up there. Also when you are landing your rover, you will need a bigger parachute, thus more weight. And if you are adding a redundant computer, that thing needs power too. And maybe even worse, your budget is a constraint. Maybe your job is to maximize reliability with respect to cost.



Redundancy > Quality

While having quality components is important, redundancy still trumps quality. As can be demonstrated with a success tree, having more of a slightly less reliable device compared to less of a more reliable device can yield better results.



Concept — Failure Tree

A failure tree is similar to a success tree, but in the complete opposite way. But instead of giving a “the computer has a .005 chance of failure”, we list what can happen, and what is the result, and how this impacts the system as a whole. So we could say “battery is leaking” and “electric current comes” would lead to “battery catches fire”, along with failure of the computer to detect the heat will finally result in “computer explodes”.

2.5 How to make a system more fault tolerant

The main point of this section can be summed up as **Redundancy**. Redundancy is the best way to achieve a more fault tolerant system. While having things such as high quality batteries, ECC memory, or well designed and implemented software, redundancy is what will let you make the most out of such equipment.

2.5.1 Redundancy in Hardware (Components)

As we have seen in modelling, having redundancy is key. But just how redundant should we make things? What is cost effective?



Concept — Triple Modular Redundancy – TMR

TMR is the standard for anything safety critical. So “triple” has an obvious meaning, but what does modular mean? This means that we are referring to modules (computer + battery), not components (computer, battery).

Although we have seen examples where modules are worse than components overall in terms of redundancy, we should remember that there are benefits to using modules.

For example, imagine we need a thermostat to measure temperature in a room. (Thermostat being the module). Sure we could have a mega thermostat that never fails because it's components are all TMR, but the downside is that our reading won't be totally accurate. What if it is next to a window in the winter? Then obviously it's readings will be lower than the room's actual temperature. But if we have three thermostats, then we can take the median, or average out, the readings for higher accuracy.



Examples of Redundancy in Hardware

1. RAID

RAID is where you have multiple drives stripped together for redundancy. But this does come with the caveat of WHAT kind of RAID you are using. RAID 0 for example, does NOT offer any redundancy benefit because of how it works.

2. Power Supplies

Its common for people to have UPS' in case their PSU poops out.

2.5.2 Redundancy in Software

Software is an interesting case, since as long as the hardware doesn't give way, software will do exactly what you tell it to do, that and only that. So much thought must go into how you make the software and verify it.

However, when you are running software, redundancy is still important. The way that redundancy in software is achieved is by running multiple copies of the software. The way that this can achieve redundancy is that if there is some mission critical variable (such as angle of attack!), after each program has calculated its value for such variable, they can vote (especially if it is discrete), or take a mean, median, mode, whatever. If they take all the values and average them out, hopefully they will smooth out any error and come to as close of an accurate value as possible.

Multiple Implementations

Given some set of requirements designed by engineers, you could have multiple teams implement the same set of requirements. By having this, your goal is to minimize the amount of faults that can occur through bugs in implementation.



Definition — N Version Programming

When you have N teams implement the same set of requirements.

Good Engineering Practices

TO ensure the best product, requirements for the software should be fixed. Additionally, they should be precise and detailed as to leave minimal room for architecture related bugs.

2.5.3 Redundancy in Information

Redundancy in information is also interesting. If we took a car for example, if the speedometer broke down, are you completely without any way of telling how fast you are going? No!



Idea — You are never without information

There is almost always some secondary source of the information you are looking for.

If your speedometer broke down (assuming you had a good reading of the last valid speed you were going at) you can estimate based on the car's acceleration or deceleration, you could use your engine RPM, you could backwards calculate based on the distance you have travelled, or you could even use your phone (GPS, maybe the phone has it's own accelerometer to tell speed). Having redundancy in information is being able to take these secondary, correlated, sources, and correctly read their data and extract the data that you want.



Idea — History Matters

In the car example, calculating speed by making use of the last known valid speed is an example of the concept that history matters.



Beware the Error!

Although good for when the main sensor that measures the quantity that you want fails, obviously it is still best to use the main sensor when possible. Doing the calculations based on other sensors have constraints. Maybe they can only tell you part of the picture. Maybe the calculation is too slow and to get the answer in a reasonable amount of time you have to approximate. Whatever the reason is, using secondary sensors introduces noise and error into your calculations.

3 Sensors

3.1 Introduction

This section will explain, what is a sensor, what kind of sensors are there, why sensors have noise and most importantly, how to correct for that noise.



Definition — Sensor

A sensor is some kind of device that is used to measure some physical quantity in the real world and digitize that value for a computing machine to use.

There are many kinds of sensors, but here are some examples:

1. LIDAR – Laser range finder, calculates time of flight of the laser to gage distance.
2. Ultra-sound and Sonar – same point as LIDAR but uses sound waves.
3. Light Sensor – Photodetector, or a camera: measures reflected light.
4. Temperature Sensor – Obvious
5. Sound – Microphone – Converts vibrations created by waves of different air densities to electrical signal.
6. Gyroscope – Measures rotation
7. Accelerometer – Measures acceleration
8. Chemical – measure properties like acidity, check for specific compounds or electric noses
9. Pressure Sensors – Detect force
10. Magnetic Sensors – Detect magnetic fields such as metal detectors or compasses
11. Location – Something like GPS.
12. Altitude – Air pressure checker or barometer
13. Humidity

14. Electricity – Measure current, voltage, static electricity, etc

Although it is a sensor's job to measure such physical quantities, no sensor is completely accurate.



Idea — Sensors are noisy

This the key concept of this section. Sensors are noisy due to a variety of reasons, and it is our job to interpret the data it provides and de-noise the best we can.

But some sensors can be less noisy than others. This is a critical component in what makes a good sensor. Other factors include:

1. Reliable
2. Sensitive to the quantity being measured
3. Not sensitive to other quantities

An example of a bad sensor for example is the PacoSensor 1000+ that measures temperature with a photo camera by checking the color of a stove top. Sure you can do that but its judging temperature through color, which is bad, we should measure as directly as possible. The PacoSensor 1000+ violates the last 2 points. It is not sensitive to the quantity being measured, and it is sensitive to other quantities.

A sensor should have a simple response curve, if we graph temp with respect to the response of the sensor, we want something linear or like exponential or log. A weird curve makes it harder to interpret the temperature from the response. $r(t) = f(t) = at + b$. But we generally want linear. This should at least be guaranteed in the sensor's operating range also called the dynamic range.

Remember, transduction is the conversion of a physical phenomenon to an electrical signal, called $s(t)$. If $f(t)$ is physical quantity, then $s(t) = f(t) + n(t)$ with $n(t)$ being noise (conversion is noisy). For a microphone, noise can include background noise, sensor distortion, electromagnetic interference, thermal noise, etc. You can account and remove some noise, but you cannot fully eliminate it.

We also have to do sampling, because $f(t)$ is continuous which computers handle that poorly. Sampling is taking a subset of a continuous curve, making discrete measurements at a uniform distribution. We then only consider those discrete points. Issues such as data loss, especially important if the continuous data changes quickly. This can be mitigated with a higher sampling rate. Also between the points we assume a straight line or constant data, which introduces distortion.

Definition – Fourier analysis is the deconstruction of a wave function into sine and cosine functions.

So we can apply Fourier analysis onto $s(t)$ to break it down into sine and cosine functions with different frequencies. The point of doing this is the find the highest frequency component that matters, (which sin/cos function contributes the most to the overall wave) because sampling has to be double the highest frequency. Otherwise you lose too much data. Human sound is around 20khz, so a microphone needs to sample at 40khz. 44.1khz is the CD sampling quality. Finding the most significant frequency also helps with filtering noise.

For us, we will look at Fourier analysis through the lens of discrete linear algebra. For this, ortho-normal basis are important.

So how does this affect us? Suppose we get 10 samples for $s(t)$. So our result is a 1x10 vector, a point in 10-d space. We can then break down our array into a linear combination of an ortho-normal basis of 10-d space. We want a useful basis, so when choosing our basis, we take our most significant sine wave, take 10 samples in the first cycle, and use that as our first vector $\hat{f}_{s1}$, then we take $\hat{f}_{s2}$ with most significant cosine wave. This is orthogonal because of properties of the sine and cosine. After that, we do 10 samples with 2 cycles in the sine wave, so on and so forth until we have 10 vectors.

Thus we can compute $s(k)$ with a linear combination of our Fourier basis. All that is left to determine is how much to scale each basis, we can compute that by doing $f_{k_i} \cdot s(k)$

In real life engineers use $e^{i\theta} = \cos \theta + i \sin \theta$.

We need to do this because the more “square wave” our thing looks, we need more sine or cosine waves, so we just take the most significant wave. We want to preserve our original signal, but that is impractical because of how much data we need, but we can get close enough only taking the most significant waves, this is an important concept in compression.

Fourier applies to all functions. The fewer waves you use, the smoother the recreated line is.

After we sample, we need to quantize the value into a storage value, in other words, how do we represent the

data? Int, double, char? The main issue here is that we can lose information due to precision issues (caused by storage limitations and data rate or how much data you need to push, an example is how many pixels need to be set for 30 fps 4k video), thus an unavoidable introduction of distortion. This kind of noise is very annoying because this kind of noise is not random, while the techniques we will learn deal with random noise. Structured noise is in general very hard to remove. If we skip quantization, we don't get this structured noise, but the issue is that the data is updated in real time by the signal, so we need a more non-volatile place to store the data.

Noise reduction/de-noising — Classical way

Best way to handle this is to make a reasonable assumption about the data.

1. Noise is IID, key being that noise is not related to each other, or the noise is not correlated (unless its structured noise).
2. Noise is zero-mean (big)
This means taking the average of many successive noise values yields something close to zero.

In practice taking these two points, for example looking at temperature. Temp usually changes very slowly. If we take a bunch of measurements really fast and take the average, we can usually recover the temperature value because noise is zero-mean.

Another example is night photography. Take multiple shots, then average them to eliminate noise.

This stops working when the signal changes too quickly. We need to sample at a rate faster than a signal is changing to use the average technique.

If you cannot average, you can try to smooth out your data using Fourier transformation. But if you need to work in real time, you can define a window of time (current time minus some fixed time) and take a weighted average using Gaussian weights. But the weighted average is an imperfect noise reduction method, and it also leads to signal loss/distortion (over-smoothing)

Something to note is that this only works for real values. But what about categorical sensors, where the signal output is some kind of discrete value or categorical value. Here, noise looks like the category returned keeps changing or reporting the wrong category. You can deal with this noise by taking the mode.